**Generational Genetic Algorithm for Number Partitioning**

Jared Kariniemi

kaja1103@stcloudstate.edu

**Abstract**

The purpose of this project is to determine how effectively an evolutionary algorithm can be used to solve the number partitioning problem. We createan evolutionary program designed for solving number partitioning as well as a non-evolutionary program to be used as a measuring stick. Testing is done on several integer sets of varying size and with varying ranges of integers. We find that neither the evolutionary algorithm nor the non-evolutionary one is more effective for all use cases.

**1. Introduction**

Number partitioning is problem that is easy to understand, but difficult to solve computationally. Optimal solutions cannot be found in all cases in better than polynomial time. We explore ways to find near optimal solutions with reasonable algorithm complexity, as well as finding optimal solutions *most* of the time in sub-polynomial time.

Evolutionary programming is a technique that is often employed to attempt to solve polynomial time problems such as number partitioning. We use this technique to design and implement an algorithm for number partitioning. A non-evolutionary algorithm is also created to which we compare the evolutionary program.

**2. Number Partitioning**

The Number Partitioning problem can be described as follows: Given a set of positive integers, divide the integers into two subsets whose sums are as close to equal as possible.

More formally, given a list of positive integers a1, a2, … , an, find a subset A {1, … , n} which minimizes the function in **figure 1**.

**Figure 1**. Number Partition function (Percus, et all, 2006).

There are no known algorithms that can find the best possible solution for the Number Partitioning problem in polynomial time, therefore the problem is classified as NP-hard.

|  |  |  |
| --- | --- | --- |
| *A* = {4, 2, 4, 7, 9} |  |  |
| List 1 | List 2 | Difference |
| {4,2,4} | {7,9} | 6 |
| {4,9} | {2,4,7} | 0 |

**Figure 2**. Two Number Partitioning solutions for set *A*.

**Figure 2** shows two solutions to the Number Partitioning problem for given set *A.* The first solution is poor since the sum discrepancy is nearly one fourth of the sum of all the numbers in the list. Better solutions can be obtained quite inexpensively. The second partition in **figure 2** has two lists with equal sums. This is a perfect partition which is the best-case solution for a partition problem number set. Not all number sets have a perfect solution. A perfect solution is always desire-able, but not the goal of an algorithm for the number partitioning problem. The goal is to find an algorithm that has manageable complexity, and acceptable discrepancy between the solution list sums.

**3. Non-Evolutionary Heuristic**

Although the problem is NP-hard, there are simple algorithms that can produce strong solutions to the problem in as little as linear time. One such Algorithm is a non-evolutionary greedy heuristic that is implemented quite simply: The given set of numbers is sorted in descending order. The sorted list is traversed, and the numbers are divided into two subsequent lists. Each number is added to whichever of the two lists has a smaller sum. A radix sort on a list of numbers can be done in O(nk) (Rowell, 2018) and the partitioning is a single run through the number set, so the reduced complexity is O(n). This algorithm provides a solution for the number set listed in **figure 2** in the manner demonstrated in **figure 3**. We refer to this algorithm as **NEA** (Non-Evolutionary Algorithm).

|  |  |
| --- | --- |
| *A* = {4, 2, 4, 7, 9} | Sorted:  *As* = {9, 7, 4, 4, 2} |
| List 1 | List 2 |
| {9} | { } |
| {9} | {7} |
| {9} | {7, 4} |
| {9, 4} | {7, 4} |
| {9, 4} | {7, 4, 2} |
| Sum = 13 | Sum = 13 |

**Figure 3**. Example execution of a non-evolutionary algorithm for number partitioning using set *A*.

Since both lists are initially empty, 9 is added to list 1, then 7 is added to list 2. 4 is added to list 2 since 7 < 9, and so on. This set happens to produce a perfect partition, but this algorithm will not always produce a perfect partition even if such a solution exists. For example, the sorted set *Bs* = {9, 7, 6, 4, 4, 2} has a perfect partition: *B1* = {9, 7} and *B2* = {6, 4, 4, 2}, but the algorithm demonstrated in **figure 3** would partition the set into *B1* = {9, 4, 4} and *B2* = {7, 6, 2}, a non-perfect partition.

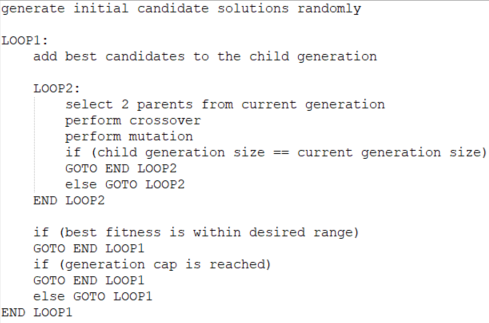
**4. Genetic Algorithm**

**4.1. Defining Genetic Algorithm**

An **evolutionary algorithm** is a search technique which emulates biological evolution to find a solution to a problem. Components of evolution such as survival of the fittest, offspring having traits of both parents, and genetic mutation are imitated in the algorithm design. A **genetic algorithm** is a type of evolutionary algorithm in which **chromosomes** are represented as strings of symbols. Offspring are created by manipulating the chromosome’s symbols in two ways: **mutation**, which is implemented by randomly changing some of the symbols, and **crossover**, which is done by taking portions of the strings from two “parents” and putting them together to form a child’s chromosome (Julstrom, 2018). Due to the complexity of finding optimal solutions to np-hard problems, genetic algorithms are a natural approach to finding acceptable solutions.

**4.2. Genetic Algorithm Structure**

This genetic algorithm, we refer to it as **JGA** (Jared’s Genetic Algorithm), creates an initial population of candidate solutions, then begins the evolutionary process. Each solution is judged by its **fitness**, a simple measurement of how far from perfect the solution is, and the solutions with the best fitness from each generation are preserved into the next generation. To create candidate solutions to fill out the rest of the next generation, two candidates from the current generation are chosen as “parents” and their chromosomes are combined using parts of the chromosome from each parent. This is the crossover step of the process. The newly created child then has a determined number of symbols in its chromosome randomly changed, which is called mutation.

 **Figure 4.** Basic structure of JGA.

Crossover and mutation are repeated to fill out the next generation, and new generations are created in this manner until a satisfactory candidate solution is found, or the maximum allowed number of generations is reached. **Figure 4** shows the basic structure of the algorithm.

**4.3. Chromosome Encoding and Evaluation**

A critical component of a genetic algorithm is how the chromosomes are encoded. The chromosome is the representation of the candidate solution, so it must be encoded in a manner that allows for easy crossover and mutation.

For number partitioning, it is sufficient to represent chromosomes using a binary string. Each 1 or 0 indicates which of the two lists the number at that position in the integer list belongs to. Crossover on a binary string is simply taking segments of the string from each parent and putting them together to form a child candidate solution. In the case of JGA, **1-point crossover** is used. A single point is randomly chosen, and bits that are in positions in the string prior to the crossover point are taken from parent 1, and the rest are taken from parent 2. Mutation on a binary string is similarly simple. One or more bits are chosen at random and flipped.

|  |  |
| --- | --- |
| *A* = {4, 2, 4, 7, 9} | Chromosome = 10011 |
| Set 1 | Set 2 |
| {4, 7, 9} | {2, 4} |
| Fitness: |(4 + 7 + 9) – (2 + 4)| = |20 – 6| = 14 | |
| Fitness %: set sum = 26 (14/26)\*100 = 54 | |

**Figure 5**. Encoding and evaluation of a chromosome for JGA

Evaluation of a candidate solution for number partitioning solutions is intuitive. The fitness of a candidate is the difference between the sums of the two lists. Lower fitness is better in this case. To normalize the value so performance can be compared across different number sets, fitness is given as a percent of the entire number set sum. **Figure 5** shows an example for how a number set is encoded as a chromosome and evaluated.

**4.4. JGA Parameters**

Population size, end condition, parent selection, and the amount of mutation done are all factors to consider when designing a genetic algorithm.

Choosing population size for genetic algorithms is an inexact science, and it is often chosen based on tested performance of the algorithm, as is the case with JGA. A population size of 50 is used, as the fitness improvements seem to flatten out between 50 and 100 population, and execution time is roughly doubled with the larger population.

The End condition for JGA can be set at any number of generations or ended when a certain fitness percentage is found. For comparing to the non-evolutionary algorithm, a cap of 50 generations is used in some test cases, while a cap of the fitness percent achieved by the non-evolutionary algorithm is used to determine the amount of generations it takes to match the algorithm’s performance.

There is a balance that needs to be found when choosing parents in a genetic algorithm. Using only the best candidates leads to getting stuck on local maxima, while choosing parents completely at random is no better than a brute force search. Selection of parents for JGA is a random selection from the top half of the candidates with 80 percent probability. So parents are selected from the entire population at random 20 percent of the time. Using the top half of candidates the majority of the time minimizes the searching in less fruitful areas of the problem space, while the occasional use of lesser candidates will prevent the local maximum issue.

Mutation is another tool that can be used to prevent the search space from becoming too narrow. We use a mutation probability of eighty percent on all newly created candidates. The number of mutations for a candidate is based on its fitness. A candidate’s fitness percentage is the percentage of bits to be mutated (flipped) and the bit positions are chosen at random. A bit may be flipped multiple times in one mutation iteration.

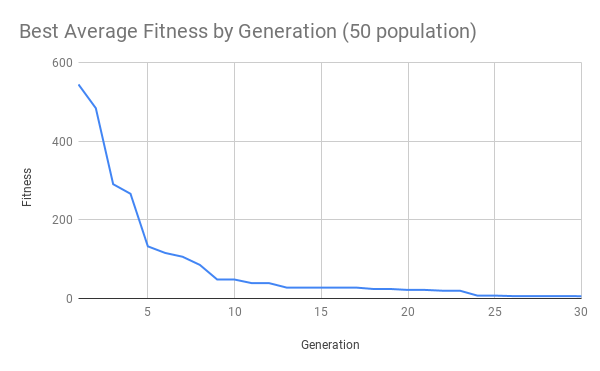
JGA also preserves the top ten percent of every generation in to the next generation with no changes. The best solutions are completely preserved, and their genetics are propagated into subsequent generations through crossover.

**5. Testing the GA**

To get a feel for the performance of JGA and how it compares to NEA, we test on a variety of number sets. JGA is run 30 times on each number set and all numbers are based on the 30 trial average.

First, we look at the average fitness produced at each generation for 30 generations on a single number set. **Figure 6** graphs the averages. This shows the progression of the algorithm’s results as generation number increases.

We compile the average generations it takes for JGA to find an optimal solution (best possible partition for the given number set), as well as the average number of generations it takes for JGA to match or beat the solution produced by NEA. This gives an idea of how the algorithm performs with a variety of number set parameters, as well as a comparison to our simple number partitioning algorithm.



**Figure 6.** Best average fitness for each generation (5 trials)

|  |  |  |  |
| --- | --- | --- | --- |
| JGA 20 Integers ranging from 1 to 100 30 trials | | | |
|  | Average | Std Dev | failed |
| Gens. to optimal | 1.2 | 2.3 | 0 |
| Gens. to match NEA | 0.2 | 0.6 | 0 |
|  | | | |
| JGA 20 Integers ranging from 50 to 100 30 trials | | | |
|  | Average | Std Dev | failed |
| Gens. to optimal | 5.8 | 8.7 | 0 |
| Gens. to match NEA | 0.3 | 0.4 | 0 |
|  | | | |
| JGA 20 Integers ranging from 1 to 1000 30 trials | | | |
|  | Average | Std Dev | failed |
| Gens. to optimal | 18.2 | 15.3 | 12 |
| Gens. to match NEA | 1.4 | 1.9 | 0 |
|  | | | |
| JGA 200 Integers ranging from 1 to 100 30 trials | | | |
|  | Average | Std Dev | failed |
| Gens. to optimal | 13.5 | 11.5 | 3 |
| Gens. to match NEA | 13.5 | 11.5 | 3 |
|  | | | |
| JGA 200 Integers ranging from 50 to 100 30 trials | | | |
|  | Average | Std Dev | failed |
| Gens. to optimal | 10.6 | 10.8 | 2 |
| Gens. to match NEA | 10.6 | 10.8 | 2 |
|  | | | |
| JGA 200 Integers ranging from 1 to 1000 30 trials | | | |
|  | Average | Std Dev | failed |
| Gens. to optimal | 17.5 | 13.5 | 19 |
| Gens. to match NEA | 10.6 | 11.1 | 0 |

**Figure 7**. Generations to match NEA performance, and to achieve an optimal solution. 6 different number sets.

**Figure 7** shows the average and standard deviation of the number of generations it took JGA to match NEA, and obtain a best possible solution for each of the 6 number sets. The failed column indicates the amount of times JGA failed to achieve an optimal solution or matching NEA’s performance in 50 generations. Number set descriptions are given at the top of each table.

**6. Observations**

In general, JGA performed well when compared to NEA on small number sets, and poorly (when execution time is considered) on larger number sets. JGA achieved an optimal solution on every number set in at worst, 11 of 30 trials. NEA achieved an optimal solution on two of the six number sets.

When comparing final solutions, JGA performs as good or better on every number set. On only 5 individual trials JGA fails to match or beat the solution provided by NEA. These were all on number sets upon which NEA found the most optimal solution. Both number sets for which NEA found optimal solutions were on larger (200) integer sets.

On smaller number sets, JGA outperforms NEA by a large margin, needing only 1.4, 0.3, and 0.2 generations on average to match the result found by NEA. There were no trials on the smaller number sets in which JGA failed to match or beat NEA’s performance.

**7. Improvements**

JGA performance was generally good, finding optimal solutions on every number set and a majority of individual trials within 50 generations. However, the non-evolutionary solution, which executes much quicker than JGA, performed almost as good or as good on larger number sets. This suggests there is room for improving the number of generations it takes the algorithm to find optimal solutions.

One possible improvement is making the population size based on the number of integers in the integer set. A series of trials can be designed to determine how population size should be related to the integer set size and range for optimal performance.

Another area that is likely improve-able is the parent selection. Using the top half of candidates on 80 percent of cases seems to work well, especially compared to JGA’s initial selection mechanism, which was completely random, but the parameter was not explored much. Comparing graphs such as the one on **figure 6** would likely reveal a more optimal selection mechanism.

**8. Conclusion**

The number partitioning problem is simple to understand, but difficult to solve computationally. To explore the best way to find adequate solutions, we created a genetic algorithm, which is often employed for np-hard problems, and a simple greedy algorithm for performance comparisons.

We found that the genetic algorithm usually will find optimal solutions within 50 generations regardless of number set size and range. The greedy algorithm found strong solutions (within 0.5 percent of total number list sum) in all cases, but only found optimal solutions in 1/3 of tested number sets.

Although the genetic algorithm found as good or better solutions in nearly all cases, the genetic algorithm is more complex, and performed only about as good as the greedy algorithm on larger number sets.

There are some areas that would likely be fruitful to explore with the aim of improving the genetic algorithm. Population size should be related to integer set size in some way for optimal performance, and parent selection is an area that could be tested further.

Based on this analysis, we cannot conclude that a genetic algorithm is objectively better than a greedy algorithm for solving number partitioning. If the goal is to find an optimal solution as quickly as possible for a number set, then this algorithm is certainly better, but if a result that falls within one percent of the total set sum is sufficient, then the greedy algorithm is more effective. If some performance improvements were made to the genetic algorithm, it is possible that it could become the better choice in most cases, but as it stands, each algorithm has areas of weakness.

**References**

Ferreira, F & Fontanari J. (1998). “Probabilistic analysis of the number partitioning problem.” *Journal of Physics A: Mathematical and General*. 31, 1.

Julstrom, B. (2018). “Evolutionary Computation” *Project 2: A Genetic Algorithm*.

Rowell, E. (2013). “*Big 0 Cheat Sheet*” Retrieved from http://bigocheatsheet.com/